



WP231 (1.1) January 6, 2006

# *HDL Coding Practices to Accelerate Design Performance*

*By: Philippe Garrault and Brian Philofsky*

---

One of the most important factors in getting the maximum performance from any FPGA design is proper coding of the design's RTL description. Certain seemingly minor decisions made while crafting an RTL-level design can mean the difference between a design operating at less than 100 MHz and one operating at more than 400 MHz.

Dependable design performance is the result of careful consideration of many factors during the design process. First, the hardware platform that best suits the design must be selected. Next, the selected device architecture and the settings and features of the implementation tools need to be studied. Lastly, and this is the purpose of this document, HDL code that maps efficiently onto the targeted device must be written. Different resources detailing each of these subjects can be found on the web. This document focuses on the latter by presenting coding styles and tips to accelerate design performance. Proper FPGA coding practices are reiterated, and the lesser known techniques directly applicable to the latest Xilinx FPGA architectures are presented.

## Use of Resets and Performance

Few system-wide choices have as profound an effect on performance, area, and power as the choice of the reset. Some system architects specify the use of a global asynchronous reset for the system for the sole purpose of circuit initialization at power-up. This is, however, not necessary for FPGA designs. With Xilinx FPGA architectures, the use of a reset and the type of reset can have serious implications on the design performance. Sub-optimal reset strategies can:

- prevent the use of a device library component, such as shift register look-up table (SRL)
- prevent the use of synchronous elements of dedicated hardware blocks
- prevent optimizations of the logic inside the fabric
- severely constrain placement and routing because reset signals often have high fanout

### SRLs

All current Xilinx FPGA architectures have the capability to configure the look-up table (LUT) element as logic, ROM/RAM, or SRL. Synthesis tools can infer the use of any one of these structures from RTL code; however, in order to use performance-optimized shift register SRL, a reset cannot be described in the code because the SRL library component does not have a reset. Using resets in code that infers shift registers requires either several flip-flops or additional logic around the SRL to allow a reset function. As illustrated in Figure 1, code without resets on shift registers generally produces a single register on the output, which is optimal for area and performance.

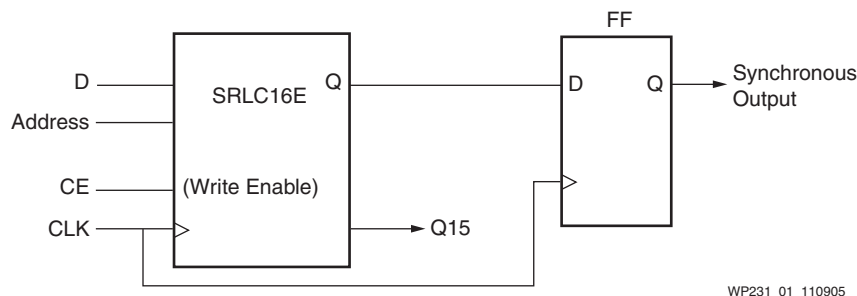


Figure 1: Performance-Optimized Shift Register

The effect on area and power is more obvious when using a reset versus not using one, but the affect on performance is a little less clear. When building a shift register out of flip-flops, performance of the shift register is generally not going to be critical because the timing path between registers (clock-to-out of a flip-flop, the associated delay in routing, and the setup time of the next flip-flop) is not normally long enough to be the longest path in the design. The added consumption of resources (flip-flops and routing), however, can have a negative influence on the placement and routing choices for other portions of the design, possibly resulting in longer routing delays for other paths in the design. In the case of adding additional logic to the SRL to emulate a reset function, a portion of this logic appears on the clock-to-out of the SRL, increasing the time it takes for the data to reach its destination logic, thus reducing performance.

#### Tips

- Avoid resets on shift registers because it prevents inference of area and performance optimized SRL library cells.

## Multipliers and RAMs

All current Xilinx FPGA architectures contain dedicated arithmetic resources. Such resources can be used to perform multiplication, as in many DSP algorithms, but can also be used in other applications, e.g., barrel shifters.

Similarly, almost every FPGA design uses RAM of various sizes, regardless of the application. All current Xilinx FPGAs contain block RAM elements that can be implemented as RAM, ROM, a large LUT, or even general logic. Using both the multipliers and RAM resources can result in more compact and higher performing designs.

The choice of the reset type can impact the design in terms of performance. Both multiplier blocks and RAM registers contain only synchronous resets; if an asynchronous reset is coded for these functions, the registers within these blocks cannot be used. This has a severe effect on performance. For example, using a fully pipelined multiplier targeting a fastest Virtex™-4 device with an asynchronous reset can result in a performance of around 200 MHz. Reworking the code to use a synchronous reset can more than double the performance to 500 MHz.

Similar to the multipliers, Virtex-4 block RAMs have optional registers. When these output registers are used, they can reduce the clock-to-out times of the RAMs and increase overall design speed. These optional registers do not have reset ports; consequently, the output registers cannot be enabled if the code describes a reset behavior.

A secondary issue arises when using the RAMs as a LUT or general logic. At times, it is advantageous for both area and performance reasons to condense several LUTs, configured as ROM or general logic, into a single block RAM. This can be done either by manually specifying these structures or automatically by constraining the synthesis tool to map portions of the logical design to unused block RAM resources. Because of the reset configuration of the block RAM, general logic can be mapped without changing the design functionality only if a synchronous reset (or no reset) is used.

### *Tips*

- Avoid asynchronous reset because it prevents packing of registers into dedicated resources and affects performance, utilization, and tool optimizations.

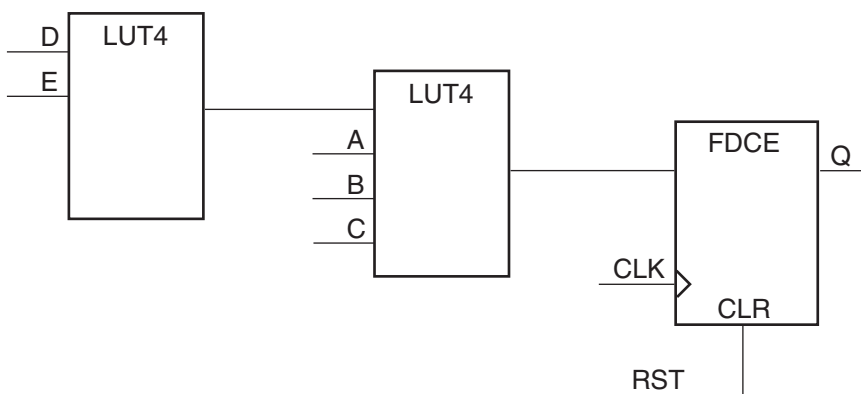
## General Logic

Asynchronous resets also have repercussions on the performance of the general logic structures. Because all Xilinx FPGA general-purpose registers contain the ability to program the set/reset as either asynchronous or synchronous, it could be perceived that there is no penalty in using asynchronous resets; that assumption is often wrong. The code examples in [Figure 2](#) illustrate how the use of asynchronous resets can inhibit optimization. If an asynchronous reset is not used, then the resources that such a signal would use are available and can be used to optimize other synchronous paths driving this register (FDRSE in [Figure 3](#)).

### **Example 1**

To implement the asynchronous reset code, the synthesis tool must infer two LUTs for the datapath because five signals are used to create this logic. A possible implementation of this code is also shown in [Figure 2](#):

VHDL	Verilog
<pre> process (CLK, RST) begin   if (RST = '1') then     Q &lt;= '0';   elsif (rising_edge(clk)) then     Q &lt;= A or (B and C and D and E);   end if; end process; </pre>	<pre> always @(posedge CLK, posedge RST)   if (RESET)     Q &lt;= 1'b0;   else     Q &lt;= A   (B &amp; C &amp; D &amp; E); </pre>



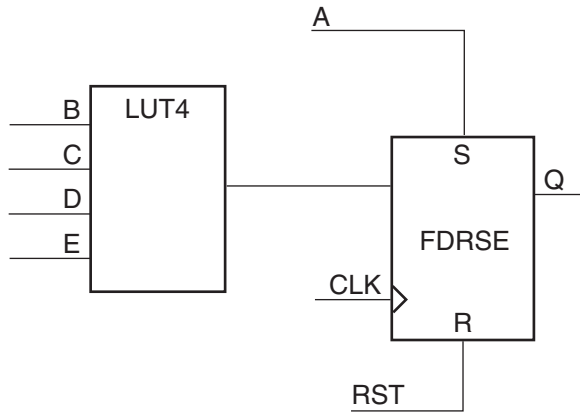
WP231\_02\_112105

Figure 2: Asynchronous Reset Implementation

**Example Two**

Rewriting this same code with a synchronous reset gives the synthesis tool more flexibility in implementing this function. A possible implementation of the code is also shown in [Figure 3](#):

VHDL	Verilog
<pre> process (CLK) begin   if (rising_edge(clk)) then     if (RST = '1') then       Q &lt;= '0';     else       Q &lt;= A or (B and C and D and E);     end if;   end if; end process;         </pre>	<pre> always @(posedge CLK)   if (RESET)     Q &lt;= 1'b0;   else     Q &lt;= A   (B &amp; C &amp; D &amp; E);         </pre>



WP231\_03\_112105

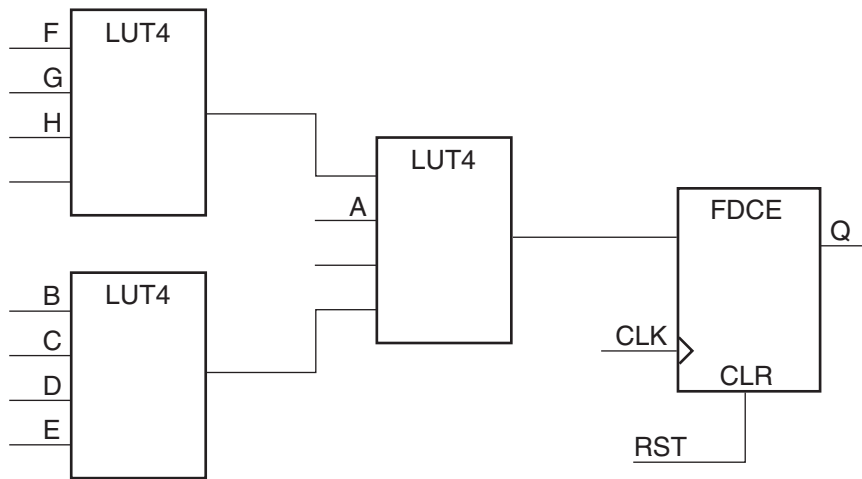
**Figure 3: Synchronous Reset Implementation Improves Performance**

With the implementation in [Figure 3](#), the synthesis tool can identify that any time A is active High, Q is always a logic 1 (the OR function). With the register (FDRSE) now configured as a synchronous set/reset operation, the set is now free to be used as part of the synchronous datapath.

**Example Three**

To further illustrate the implications of asynchronous set or reset on performance, a more complex function, with 8 contributing signals, can be examined. To implement this function, a minimum of 3 LUTs is needed. A possible implementation of the code is shown in [Figure 4](#):

VHDL	Verilog
<pre> process (CLK, RST) begin   if (RST = '1') then     Q &lt;= '0';   elsif (rising_edge(clk)) then     Q &lt;= (F or G or H) and           (A or (B and C and D and E));   end if; end process; </pre>	<pre> always @(posedge CLK, posedge RST)   if (RESET)     Q &lt;= 1'b0;   else     Q &lt;= (F   G   H) &amp;           (A   (B &amp; C &amp; D &amp; E)); </pre>



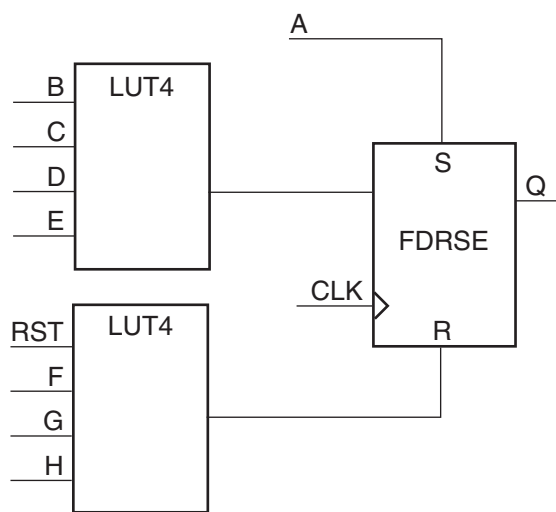
WP231\_04\_112105

Figure 4: Asynchronous Reset Implementation for an Eight-Input Function

**Example Four**

Figure 5 shows the same code written with a synchronous reset; a possible implementation of the code is also provided.

VHDL	Verilog
<pre> process (CLK) begin   if (rising_edge(clk)) then     if (RST = '1') then       Q &lt;= '0';     else       Q &lt;= (F or G or H) and             (A or (B and C and D and E));     end if;   end if; end process; </pre>	<pre> always @(posedge CLK)   if (RESET)     Q &lt;= 1'b0;   else     Q &lt;= (F   G   H) &amp;           (A   (B &amp; C &amp; D &amp; E)); </pre>



WP231\_05\_112105

Figure 5: **Synchronous Control Signal Reduces Number of Logic Levels**

Again, the resulting implementation in [Figure 5](#) not only uses fewer LUTs to implement the same logic function but can also potentially result in a faster design due to the reduction of logic levels for practically every signal creating this function. Because most of the logic in a design is synchronous, using synchronous or no reset at all allows for further design optimizations, reduced area, and optimal performance.

#### *Tips*

- Check whether a global reset is needed.
- Avoid asynchronous control signals.

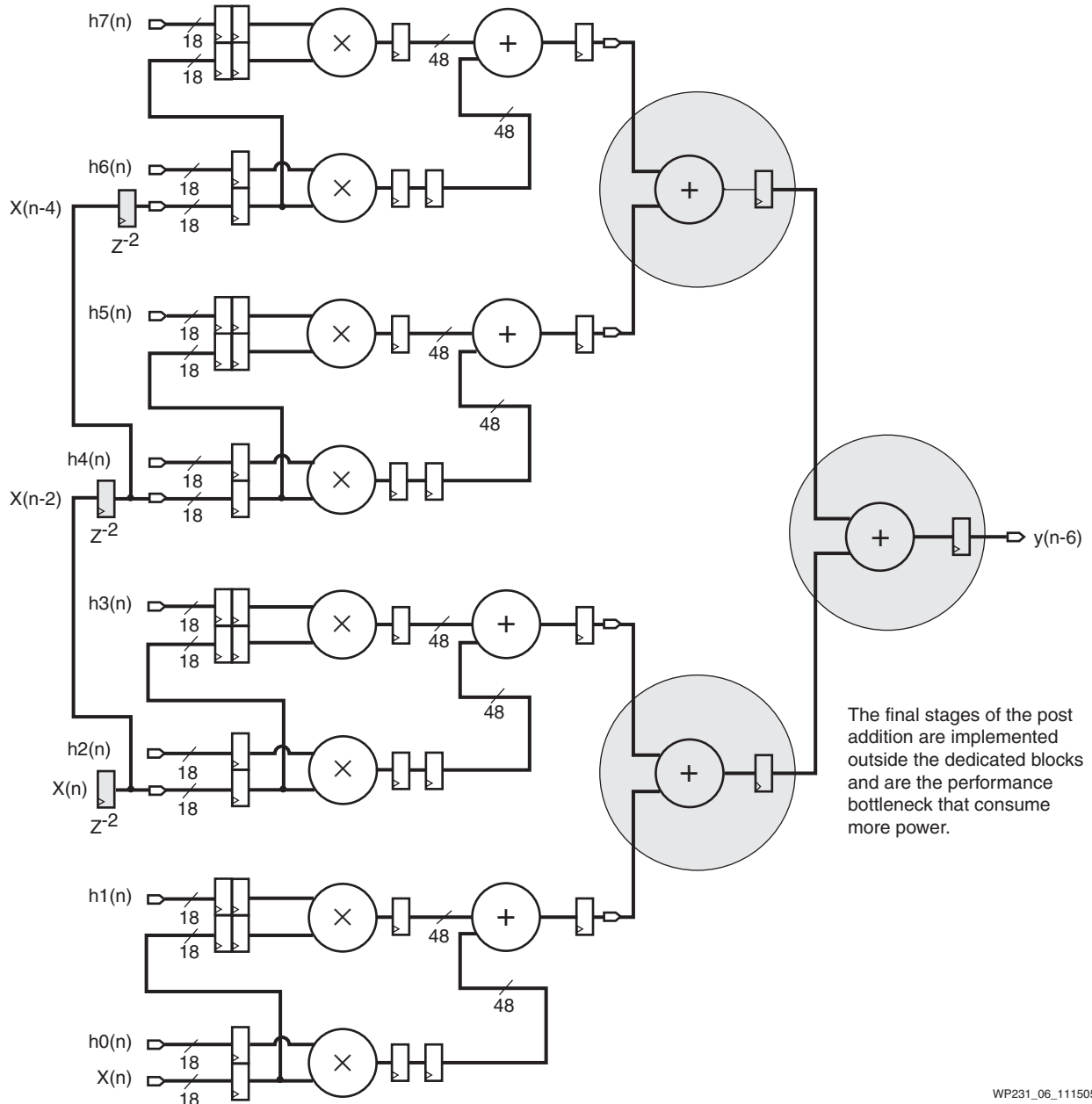
### Use Adder Chains Instead of Adder Trees

Many signal processing algorithms perform an arithmetic operation on an input stream of samples followed by a summation of all outputs of the arithmetic operation. To implement the summation in parallel architectures, such as FPGAs, the adder tree structure is typically used. One difficulty with the adder tree concept is the varying nature of its size. The number of adders depends on the number of inputs in the adder tree. The more inputs in the adder tree, the more adders that are needed, increasing both the number of logic resources and power consumption. Larger trees also mean larger adders in the last stages of the tree; large adders further reduce system

performance by increasing the number of logic levels. An approach to minimize device utilization and power consumption while maintaining the high performance of adder trees is to implement the adder trees as dedicated silicon resources (see [Figure 6](#)). However, it is impossible for an FPGA manufacturer to find a structure that allows both the majority of adder trees to be implemented in a dedicated resource while keeping silicon area to a minimum.

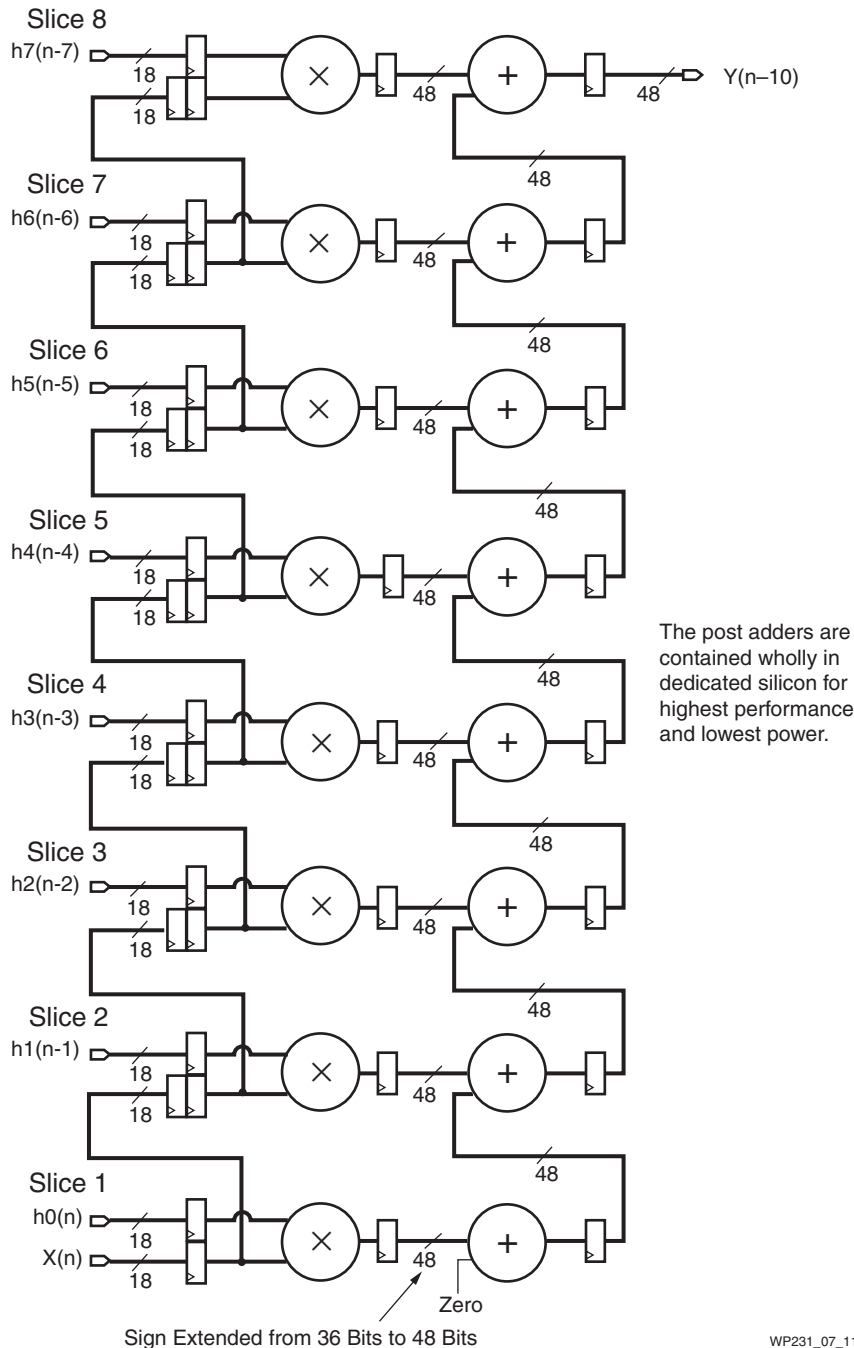
With its columns of DSP48 dedicated silicon, the Virtex-4 family has a different approach in implementing summations. As illustrated in [Figure 7](#), it involves computing the summation incrementally using chained adders instead of adder trees. No other FPGA uses this approach. It is key to maximizing performance and lowering power for DSP algorithms because both logic and interconnect are contained entirely within the dedicated silicon. When pipelined, the performance of the DSP48 block is 500 MHz in the fastest speedgrade, independent of the number of adders. Cascading ports combined with the 48-bit resolution of the adder/accumulator allow the current sample calculation along with the summation of all computed samples thus far.





WP231\_06\_111505

Figure 6: Adder Tree Structure Reduces Performance and Consumes More Power



**Figure 7: Chaining Adders Provide Predictable Performance**

To take advantage of the Virtex-4 adder chain structure, the adder tree description is replaced with an adder chain description. This process of converting a direct form filter to a transposed or systolic form is detailed in [UG073: XtremeDSP for Virtex-4 FPGAs User Guide](#) and typically involves adding latency to the design. After the conversion is completed, the algorithm can run much faster than the application requirements. In that case, the device utilization and power consumption can be further reduced by using either multichanneling or folding techniques. Both techniques help implement designs in smaller devices or allow functionality to be added to a design using the freed resources.

- Multichanneling is a process that leverages very fast math elements across multiple input streams (channels) with much lower sample rates. This technique increases silicon efficiency by a factor almost equal to the number of channels. Multichannel filtering can be viewed as time-multiplexing single-channel filters. For example, in a typical multichannel filtering scenario, multiple input channels are filtered using a separate digital filter for each channel. Taking advantage of the Virtex-4 DSP48 slice, a single digital filter can filter all eight input channels by clocking the single filter with an 8x clock. This reduces the number of FPGA resources needed by almost eight times.
- Folding is a similar concept. Instead of time-multiplexing several input streams, the taps of a single filter are time multiplexed, allowing the filter to use fewer resources. The freed resources can be used for other purposes.

#### *Tips*

- Use the Virtex-4 adder cascade descriptions in lieu of adder trees.

### **Maximize Block RAM Performance**

When it comes to efficiently inferring memory elements, several factors affecting performance must be considered:

- deciding to use dedicated blocks or distributed RAMs
- using the output pipeline register
- avoiding asynchronous resets

Other factors, namely HDL coding style and synthesis tool settings, can substantially impact memory performance.

### **HDL Coding Style**

When inferring dual port block memories, it is possible for both ports to access the same memory cell at the same time. For example, if both ports are simultaneously writing different values at the same memory cell, this creates a collision, and the memory cell content cannot be guaranteed.

Another common memory configuration example is when the value on the output of the memory depends on the target device. The latest Virtex and Spartan™ families have three programmable operating modes to govern the memory output while a write operation is occurring. Additional information about these operating modes is provided in the device user guides.

#### **Example 5**

As illustrated in [Table 1](#), synthesis tools are able to infer either of these modes depending on the coding style.

Table 1: Block RAM Operating Mode Inference Example

VHDL	Verilog
<pre>-- 'write first' or transparent mode process (clk) begin     if (rising_edge(clk)) then         if (we = '1') then             mem(conv_integer(addr)) &lt;= di ;             do &lt;= di;         else             do &lt;= mem(conv_integer(addr));         end if;     end if; end process;</pre>	<pre>// 'write first' or transparent mode always @(posedge clk) begin     if(we) begin         do &lt;= data;         mem[address] &lt;= data;     end else         do &lt;= mem[address];     end end</pre>
<pre>-- 'read first' or read before write(slower) process (clk) begin     if (rising_edge(clk)) then         if (we = '1') then             mem(conv_integer(addr)) &lt;= di;         end if;         do &lt;= mem(conv_integer(addr));     end if; end process;</pre>	<pre>// 'read first' or read before write mode(slower) always @(posedge clk) begin     if (we)         mem[address] &lt;= data;     do &lt;= mem[address]; end</pre>
<pre>-- 'no change' mode process (clk) begin     if (rising_edge(clk)) then         if (we = '1') then             mem(conv_integer(addr)) &lt;= di ;         else             do &lt;= mem(conv_integer(addr));         end if;     end if; end process;</pre>	<pre>// 'no change' mode always @(posedge clk)     if (we)         mem[address] &lt;= data;     else         do &lt;= mem[address]; end</pre>

### Tips

- Avoid "read before write" mode to achieve maximum block RAM performance.

### Synthesis Tool Settings

Another important factor that can significantly affect the block memory performance is the synthesis tool settings. Some synthesis tools (such as Synplify<sup>®</sup> from Synplicity<sup>®</sup>) insert bypass logic around the RAM to prevent the possible mismatch between RTL and hardware behavior. This extra logic is meant to force the RAM outputs to some known values when read and write operations occur on the same memory cell. If the designer knows that simultaneous read and write operations on the same memory cell will never happen, the synthesis tool settings can be used to prevent bypass logic from being added to the application (see [Table 2](#)). Extra logic has a negative impact on the memory performance because it adds overhead logic on all output paths of the memory. Eliminating or preventing the additional logic maintains the memory performance.

Table 2: Disabling Insertion of Conflict Avoidance Glue Logic

VHDL	Verilog
<pre> type mem_type is array (127 downto 0) of     std_logic_vector (7 downto 0); signal mem : mem_type;  -- disable conflict avoidance logic attribute syn_ramstyle of mem : signal is     "no_rw_check"; </pre>	<pre> // disable conflict avoidance logic reg [7:0] mem [127:0] /* synthesis     syn_ramstyle=no_rw_check*/; </pre>

### Tips

- Check synthesis tools settings, inference templates, and limitations to maximize memory block's performance.

## General Use of Registers

FPGA architectures have one register for every LUT, with additional registers available in I/Os and dedicated blocks, such as memory and DSP elements. Using these resources is important to achieve maximum performance. Registers can serve multiple purposes for accelerating design performance. They can be used to reduce the number of logic levels in critical paths, critical nets fanout, set-up, and clock-to-out of I/Os or dedicated blocks. Guidelines to optimize the available registers appear in the following subsections.

### Use Dedicated Block Registers

FPGAs have dedicated circuitry for functions used in most designs, such as memory or DSP blocks. These blocks have optional registers. Enabling these registers accelerates the block performance by reducing set-up, clock-to-out and/or increasing block clock speed. Synthesis tools automatically try to pack registers into these blocks because it saves area, improves power, and provides the best performance. The designer can use constraints to control synthesis tool mapping of inferred components. When instantiating these blocks, however, the set of registers that give the best performance for the application need to be enabled; by default, the synthesis tools do not optimize user-instantiated architecture components. Under the best circumstances, all registers should be enabled for maximum performance; however, latency requirements might not always allow this. In these cases, the responsibility falls onto the designer to enable the right set of registers. For example, when only the multiplier (MREG) and the input registers (AREG and BREG) are used, then the Virtex-4 DSP48 cell (fast speedgrade) has a setup of 1.8 ns and 2.3 ns clock-to-output. Maintaining the same functionality and latency but enabling the accumulator output register (PREG) instead of the MREG, the set-up time increases to 3 ns while the clock-to-out shrinks to 0.6 ns. See [Figure 8](#).

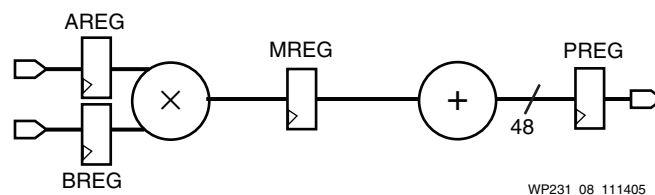


Figure 8: DSP48 Internal Registers

It is, therefore, important to consider:

- the timing of logic driving or being driven by dedicated blocks
- the positioning of registers in the HDL code

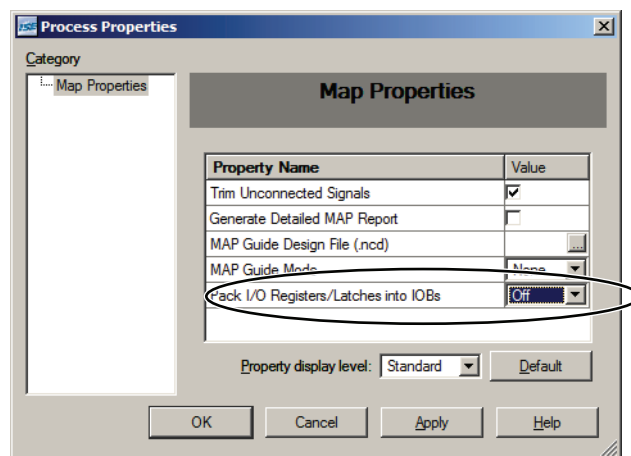
#### Tips

- When inferring dedicated blocks, synthesis constraints should be applied if the synthesis tool does not enable the best set of registers for performance. (Instantiation can also be considered).
- When selecting which register to enable for instantiated components, ensure the maximum number of registers are used and take into account delays to, from, and between registers within the block.

### Use of I/O Registers

All Xilinx FPGAs contain dedicated registers on the FPGA input and output paths. By utilizing these registers, set-up times for the input paths and clock-to-out times for the output paths can be minimized, making it much easier to meet timing requirements for capturing and providing data to external devices. At times, however, using the dedicated I/O registers can have a negative effect on meeting the timing within the FPGA; their use can lengthen route delays to the internal logic. These registers should be placed into the I/O when it is necessary to meet I/O timing, or they should be placed within the FPGA fabric when the I/O timing requirement permits it. Some synthesis tools, like Synplify, automatically place registers in either the fabric or the I/O depending on timing specifications. If the synthesis tool does not support automatic placement or if manual control of register placement is desired, the following steps must be performed:

1. Disable any global I/O register placement options for the synthesis tool (refer to synthesis tool documentation).
2. Specify whether the register should be placed into the I/O by adding an IOB=TRUE in the UCF file or source HDL code (refer to the Constraints Guide for details on the IOB constraint).
3. Disable the Map option "Pack I/O Registers/Latches into IOBs" in ISE Project Navigator (or do not use the -pr switch if running from command-line). This disables automatic pushing of registers into the I/O. See [Figure 9](#).



WP231\_09\_110905

Figure 9: Disable Indiscriminate Packing of Registers in the I/O Cells

Controlling I/O register usage balances the timing for datapaths entering and exiting the FPGA with the need to meet timing specifications inside the FPGA. Another notable guideline is to describe registers on all input and output ports to the FPGA on the top-level of the HDL code. Specifying the registers on the top-level hierarchy of the code avoids placement conflicts when using hierarchical design methods for implementing the FPGA. It also avoids creating hierarchical names for the port description which some board schematic capture tools do not accept.

#### Tips

- Disable global packing of registers into I/O cells. Instead, only constrain registers for which timing is critical on the printed circuit board to be packed into the FPGA I/O cell.

### Replicate Registers with High-Fanout

Register replication is a technique used to increase the speed of critical paths by making copies of registers to reduce the fan-out of a given signal. This gives the implementation tools more leeway to place and route the different loads and associated logic. Synthesis tools use this technique extensively. If in the timing report high fanout nets with long route delays are reported as critical paths, replication constraints on the synthesis tool or manual replication registers should be considered. The HDL code in [Table 3](#) illustrates how to manually replicate a 64-load signal one time.

**Table 3: Register Replication Example**

VHDL	Verilog
<pre>attribute EQUIVALENT_REGISTER_REMOVAL : string; attribute EQUIVALENT_REGISTER_REMOVAL of     ce1 : signal is "NO"; attribute EQUIVALENT_REGISTER_REMOVAL of     ce2 : signal is "NO";  begin  -- Clock enable register with 64 fanout -- replicated once process (clk) begin     if (rising_edge(clk)) then         ce1 &lt;= ce;         ce2 &lt;= ce;     end if; end process;  process (clk) begin     if (rising_edge(clk)) then         if (ce1='1') then             res(31 downto 0) &lt;= a_data(31 downto 0);         end if;         if (ce2='1') then             res(63 downto 32) &lt;= a_data(63 downto 32);         end if;     end if; end process;</pre>	<pre>(*EQUIVALENT_REGISTER_REMOVAL="NO"*) reg ce1,     ce2;  // Clock enable register with 64 fanout // replicated once always @(posedge clk) begin     ce1 = ce;     ce2 = ce; end  always @(posedge clk) begin     if (ce1)         res[31:0] &lt;= a_data[31:0];     if (ce2)         res[63:32] &lt;= a_data[63:32]; end</pre>

### Example 6

Many times an additional synthesis constraint needs to be added to ensure that a manually duplicated register is not optimized away by the synthesis tool. In the above example, the XST syntax was used (EQUIVALENT\_REGISTER\_REMOVAL).

Most synthesis tools use a fanout threshold limit to automatically determine whether a register should be duplicated or not. Adjusting this global threshold allows for automatic duplication of high fanout nets but does not offer a finer level of user control as to which specific registers can be duplicated. A better recourse is to apply attributes on specific registers or levels of hierarchy to specify which registers can or cannot be replicated.

#### Tips

- If the place-and-route tool reports that high-fanout signals are limiting the design performance, consider replicating them.

### Add Levels of Pipeline

Another way to increase performance is to restructure long datapaths with several levels of logic and distribute them over multiple clock cycles. This method allows for a faster clock cycle and increased data throughput at the expense of latency and pipeline overhead logic management. Because FPGAs are register-rich, the additional registers and overhead logic are usually not an issue. With this technique, the datapath spans multiple cycles; therefore, special considerations must be used for the rest of the design to account for the added path latency. [Table 4](#) presents a coding style that adds six levels of registers on the output of a 32x32 multiplier. The synthesis tool pipelines these registers into the optional registers of the Virtex-4 DSP48 to maximize data throughput.

Table 4: Inserting Levels of Pipeline

VHDL	Verilog
<pre> type regbank6x64 is array(PIPE-1 downto 0) of     std_logic_vector(63 downto 0); signal prod: regbank6x64;  -- 32x32 multiplier with 4 DSP48 (PIPE=6) prod(0) &lt;= a * b;  regbank: for i in 1 to PIPE generate begin     process (clk) begin         if (rising_edge(clk)) then             prod(i) &lt;= prod(i-1);         end if;     end process; end generate;  mult_out &lt;= prod(PIPE);                 </pre>	<pre> parameter PIPE = 6; reg signed [63:0] prod [PIPE-1:0];  // 32x32 multiplier with 4 DSP48 (PIPE=6) always @(posedge clk) begin     prod[0] &lt;= a * b;     for (i=1; i&lt;=PIPE-1; i=i+1)         prod[i] &lt;= prod[i-1];     end assign mult_out = prod[PIPE-1];                 </pre>

If the design verification methodology and toolset permit, then combining retiming with pipelining to further enhance the design performance should be considered. Retiming is a synthesis or place-and-route algorithm that automatically moves registers (register balancing) across combinatorial logic to improve timing while ensuring identical behavior, as seen from the primary inputs and outputs of the design.



Retiming simplifies RTL design because performance can be improved without any code changes. However, retiming makes design verification somewhat more complicated because the register names, position, and functionality no longer match the RTL description. Therefore, some designers avoid using retiming. When retiming is not an option, knowledge about the device helps the designer to describe registers in the RTL so that these registers map efficiently onto the available device resources. Logic implementation in the underlying architecture, in terms of performance, should always be considered. The number of logic levels and possible signal fanouts each particular piece of code might create should also be considered. A balanced placement of registers in the RTL code must be chosen so that no portion of the design has a significantly larger portion of the logic levels or fanout. By following these guidelines, designs should reach their full performance potential for a given mapping of logic.

#### *Tips*

- Improve design performance by balancing the number of logic levels between registers. Add levels of pipeline in the RTL code, apply the synthesis tool's retiming option whenever relevant, or both.

### **Inference vs. Instantiation**

It usually is preferable to behaviorally describe the design and let the synthesis tool do the mapping of the code into the gates available in the FPGA. In addition to making the code more portable, all inferred logic is visible to the synthesis tool, allowing the tool to perform optimizations between functions. These optimizations include: logic replications, restructuring and merging, or retiming to balance logic delay between registers. When device library cells are instantiated, synthesis tools do not optimize them by default. Even when instructed to optimize the device library cells, synthesis tools generally cannot perform the same level of optimization as with the RTL. Therefore, synthesis tools typically only perform optimizations on the paths to and from these cells but not through the cells. For example, if an SRL is instantiated and the cone of logic driven by this SRL is long, then this path might become a bottleneck (mentioned in the [SRLs](#) section). The SRL has a longer clock-to-out delay than a regular register. To preserve the area reduction provided by the SRL while improving its clock-to-out performance, an SRL of one delay less than the actual desired delay is created, with the last stage implemented in a regular flip-flop.

There are, however, cases where instantiation is desirable. This is typically when the synthesis tool mapping does not meet the timing, power, or area constraints, or if a particular feature within an FPGA is not able to be inferred. With instantiation, the designer has total control over the synthesis tool. For instance, to achieve better performance, the designer can implement a comparator using only LUTs, instead of the combination of LUT and carry chain elements usually chosen by the synthesis tool. In other instances, instantiation is the only way to make use of the complex resources available in the device. This can be due to:

- HDL language restrictions. For instance, it is not possible to describe double data rate (DDR) outputs in VHDL because it requires two separate processes to drive the same signal.
- Hardware complexity. It is easier to instantiate Virtex-4 I/O SerDes elements than it is to behaviorally describe them.
- Synthesis tools inference limitations. For instance, synthesis tools currently do not have the capability to infer the Virtex-4 FIFOs or the DSP48 symmetric rounding and saturation from behavioral descriptions. Therefore, none of this circuitry is used unless the user instantiates it.

### Tips

- Try to describe the maximum of your design behaviorally.
- When synthesized behavioral code does not meet requirements, review the synthesis tool timing constraints and settings before replacing the code with device library component instantiations.
- Consider the "ISE Language Templates" when writing common Verilog and VHDL behavioral constructs.
- Consider the CORE Generator™, architecture wizard, or ISE language templates when instantiating device library components.

## Clock Enable and Gated Clocks

For CLB registers, Xilinx generally recommends using the dedicated clock-enable port instead of gating the clock port. Gated clocks can cause glitches, increased clock delay, clock skew, and other undesirable effects. Using clock enable saves clock resources and can improve timing characteristics and analysis of the design. There are several ways to use clock-enable resources available on devices. To gate entire clock domains for power reduction, it is preferable to use the clock-enabled global buffer resource called BUFGCE (see [Figure 10](#)).

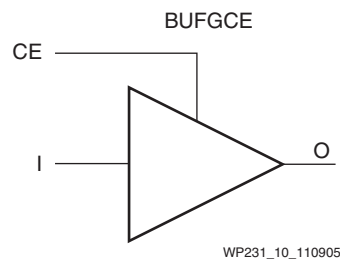
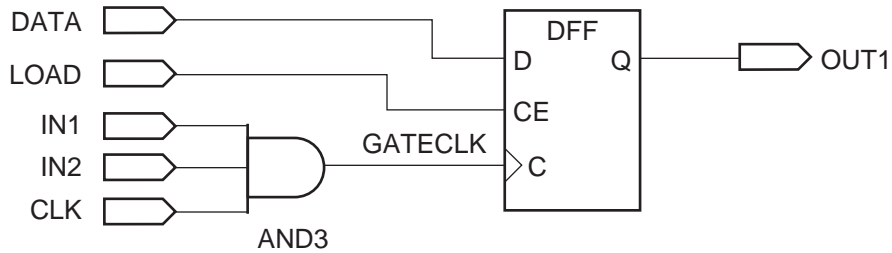


Figure 10: **BUFGCE**

For applications that only attempt to pause the clock for a few cycles on small areas of the design, the preferred method is to use the clock-enable pin of the FPGA register. The first example (see [Figure 11](#)) in this section illustrates an inefficient way of gating clock signals, while the second example (see [Figure 12](#)) shows a modified version of the code that maps efficiently into the clock-enable pin.

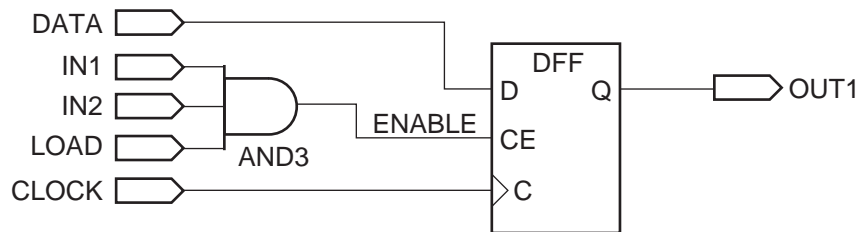
VHDL	Verilog
<pre>GATECLK &lt;= (IN1 and IN2 and CLK); process (GATECLK) begin   if (rising_edge(GATECLK)) then     if (LOAD = '1') then       OUT1 &lt;= DATA;     end if;   end if; end process;</pre>	<pre>assign GATECLK = (IN1 &amp; IN2 &amp; CLK); always @(posedge GATECLK) begin   if (LOAD)     OUT1 &lt;= DATA; end</pre>



WP231\_11\_112105

Figure 11: Gated Clock – Xilinx Does \*Not\* Suggest Using This Coding Style

VHDL	Verilog
<pre>ENABLE &lt;= IN1 and IN2 and LOAD; process begin   if (rising_edge(CLOCK)) then     if (ENABLE = '1') then       DOUT &lt;= DATA;     end if;   end if; end process;</pre>	<pre>assign ENABLE = (IN1 &amp; IN2 &amp; LOAD); always @(posedge CLOCK) begin   if (ENABLE)     DOUT &lt;= DATA; end</pre>



WP231\_12\_112105

Figure 12: Clock Enable - Efficient Way of Gating a Clock Signal

### Tips

- Avoid gated clocks.
- Use the clock-enable port of global clock buffers to stop the clock on entire clock domains.
- Use the clock-enable port of registers to locally disable the clock.
- Consider replicating the clock-enable signal if it appears to be part of the paths that do not meet your timing requirements.

## Nested If-Then-Else, Case Statements, and Combinatorial For-Loops

Nested *if* and *case* statements and statements inside of other statements should be avoided. The number of nests in the code should be minimal. Too many *if* statements inside of other *if* statements make the line length too long and can inhibit synthesis optimizations. By keeping the nested statements to a minimum, the code is generally more readable, more portable, and can be more easily formatted for printing.

When describing *for-loops* in HDL, it is preferable to place at least a register in the datapath especially when there are arithmetic or other logic intensive operations. During compilation, the synthesis tool unrolls the loop. Without these synchronous elements, the synthesis tool concatenates the logic created at each iteration of the loop, potentially resulting in very long combinatorial paths and limiting design performance.

### Tips

- Minimize the number of nests within sequential statements.
- Registers should be added to *for-loops* that are generating long combinatorial paths.

## Hierarchy

Selection of the design hierarchy partitioning is often driven by the ease of writing the HDL code. However, to achieve the best overall design hierarchy in terms of performance while reducing the overall time to complete the design, it is often better to consider design optimization, implementation methodologies, and verification. In terms of optimization, most synthesis tools treat logical design hierarchy as "soft," meaning they preserve hierarchy where possible but dissolve hierarchy or modify the footprint and logic contents within hierarchy where it allows for optimizations. If, however, hierarchical design practices like Incremental Design or KEEP HIERARCHY are employed for verification purposes, then optimizations cannot be performed across logical boundaries. If certain guidelines are not properly followed, less optimal designs with more logic levels or more placement restrictions can result. Even if non-hierarchical design implementation flows are used, these guidelines make it easier and, thus, more likely that the synthesis and place-and-route tools make the best choices in terms of logic optimization and placement. The following list briefly explains these guidelines:

1. Register all inputs and outputs to a given hierarchy. If that is not possible, always at least register the outputs. This ensures that the hierarchy will not interrupt general logic optimization.
2. Place all I/O components including any instantiated I/O buffers, registers, DDR circuitry, SerDes, or delay elements on the top-level of the design. If it is not possible to place them on the top-level, ensure that they are all contained within a single hierarchy.

3. Any register or logic sets that need to be placed within a common FPGA feature or resource should be contained in the same hierarchy level. For instance, if the design requires a multiplier, accumulator, and associated registers placed into a Virtex-4 DSP48, all of those elements should be contained within a single hierarchy module.
4. Any logic in which the synthesis tool employs resource sharing should be contained within the same hierarchy.
5. Manually duplicate registers with high fanouts at hierarchy boundaries.

By following these simple guidelines, it is far less likely that the chosen hierarchy will interfere with design optimization and performance. If at any time these rules must be broken, it is encouraged to investigate the critical paths in the implemented design to determine if modifying the hierarchy might have an impact on the end performance of the design.

---

## Conclusion

Recent advances in synthesis and place-and-route algorithms have made achieving the best performance out of a particular device much more straightforward. Synthesis tools are able to infer and map complex arithmetic and memory descriptions onto the dedicated hardware blocks. They also perform optimizations such as retiming and logic and register replications. Based on timing constraints, the place-and-route tool can now restructure the netlist and perform timing-driven packing and placement to minimize placement and routing congestions. However, given a particular RTL description, there is only so much the tools can do to maximize performance. If more performance is needed in a design, then a very efficient way to proceed is by learning more about the target device, adjusting the tools constraints and options, and using the coding guidelines illustrated in this document.

---

## Additional Resources

These resources provide additional tips on how to maximize design performance:

Synthesis and Verification Design Guide:

[http://www.xilinx.com/support/software\\_manuals.htm](http://www.xilinx.com/support/software_manuals.htm)

Xilinx TechXclusives articles:

[http://www.xilinx.com/xlnx/xweb/xil\\_tx\\_home.jsp](http://www.xilinx.com/xlnx/xweb/xil_tx_home.jsp)

XST User Guide:

[http://www.xilinx.com/support/software\\_manuals.htm](http://www.xilinx.com/support/software_manuals.htm)

Verilog coding style and papers from Sunburst Design:

<http://www.sunburst-design.com/papers/>

Efficient coding style for state machines:

[http://www.xilinx.com/xcell/xl38/xcell38\\_24.pdf](http://www.xilinx.com/xcell/xl38/xcell38_24.pdf)

---

## Revision History

The following table shows the revision history for this document.

Date	Version	Revision
12/05/05	1.0	Initial Xilinx release.
01/06/06	1.1	Typographical edits.